Puma - A Generator for the Transformation of Attributed Trees

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

ACHERN, GERMANY

Cocktail

Toolbox for Compiler Construction

Puma - A Generator for the Transformation of Attributed Trees

Josef Grosch

Sept. 25, 2008

Document No. 26

Copyright © 2008 Dr. Josef Grosch

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Höhenweg 6
77855 Achern
Germany

Phone: +49-7841-669144
Fax: +49-7841-669145
Email: grosch@cocolab.com

Puma - A Generator for the Transformation of Attributed Trees

1. Introduction

Puma is a tool supporting the transformation and manipulation of attributed trees. It is based on pattern-matching, unification, and recursion. *Puma* cooperates with the generator for abstract syntax trees *ast* [Groa], which already supports the definition, creation, and storage of attributed trees. *Puma* adds a concise notation for the analysis and synthesis of trees. The pattern-matching capability facilitates the specification of decision tables. *Puma* provides implicit declaration of variables, strong type checking with respect to trees, and checks the single assignment restriction for variables. The output is the source code of a program module written in one of the target languages C, C++, Java, or Modula-2. This module implements the specified transformation routines. It can be integrated easily with arbitrary program code. The generated routines are optimized with respect to common subexpression elimination and tail recursion.

The intended use of this tool proceeds in three steps: First, a tree is constructed either by a parser, a previous transformation phase, or whatever is appropriate. Second, the attributes in the tree are evaluated either using an attribute grammar based tool, by a *puma* specified tree traversal and attribute computations, or by hand-written code. Third, the attributed tree is transformed or mapped to another data structure by a *puma* generated transformation module. These steps can be executed one after the other or more or less simultaneously. Besides trees, *puma* can handle attributed graphs as well, even cyclic ones. Of course the cycles have to be detected in order to avoid infinite loops. A possible solution uses attributes as marks for nodes already visited.

A transformer module can make use of attributes in the following ways: If attribute values have been computed by a preceding attribute evaluator and are accessed in read only mode then this corresponds to the three step model explained above. A *puma* generated module can also evaluate attributes on its own. A further possibility is that an attribute evaluator can call *puma* subroutines in order to compute attributes. This is especially of interest when attributes depend on tree-valued arguments.

The tool supports two classes of tree transformations: *mappings* and *modifications*. Tree mappings map an input tree to arbitrary output data. The input tree is accessed in read only mode and left unchanged. Tree *modifications* change a tree by e. g. computing and storing attributes at tree nodes or by changing the tree structure. In this case the tree data structure serves as input as well as output and it is accessed in read and write mode.

The first class covers applications like the generation of intermediate languages or machine code. Trees are mapped to arbitrary output like source code, assembly code, binary machine code, linearized intermediate languages like P-Code, or another tree structure. A further variant of mapping is to emit a sequence of procedure calls which are handled by an abstract data type.

The second class covers applications like semantic analysis or optimization. Trees are decorated with attribute values, properties of the trees corresponding to context conditions are checked, or trees are changed in order to reflect optimizing transformations.

The contents of this manual is organized as follows: Section 2 gives an overview and describes the cooperation of *puma* and *ast*. Section 3 describes the specification language of *puma*. Section 4 describes the output of *puma*. Section 5 contains the UNIX manual page. Appendix 1 contains the syntax summary. Appendix 2 presents an example from a compiler for MiniLAX. Appendix 3 lists the type specific equality operations for the target languages C, C++, Java, and Modula-2.

2. Overview

The input of a transformer is a tree which might be decorated with attributes. The structure of the legal input trees and the desired transformation are described in two separate documents. Both documents are processed by the separate tools *ast* and *puma*. The cooperation between those tools is depicted in Figure 1. The structure of the trees including their attributes is described by a tree grammar and is fed into *ast*. *Ast* produces the source code of a module that defines, stores, and manipulates the specified tree and an internal description of the tree in the file *Tree.TS*. This file and the description of the intended transformation are the input of *puma*. *Puma* generates a module that implements the specified transformation by a set of subprograms which use the tree module produced by *ast*. The two generated modules, which are named *Tree* and *Trafo* by default, consist of two files: The header, interface, or definition part and the implementation part. Both modules must be compiled and linked, eventually with other modules, to yield an executable program.

For the following we assume the reader to be familiar with the tool *ast*. *Ast*'s input language is used to define the node types, the subtype relation between the node types, and the children and

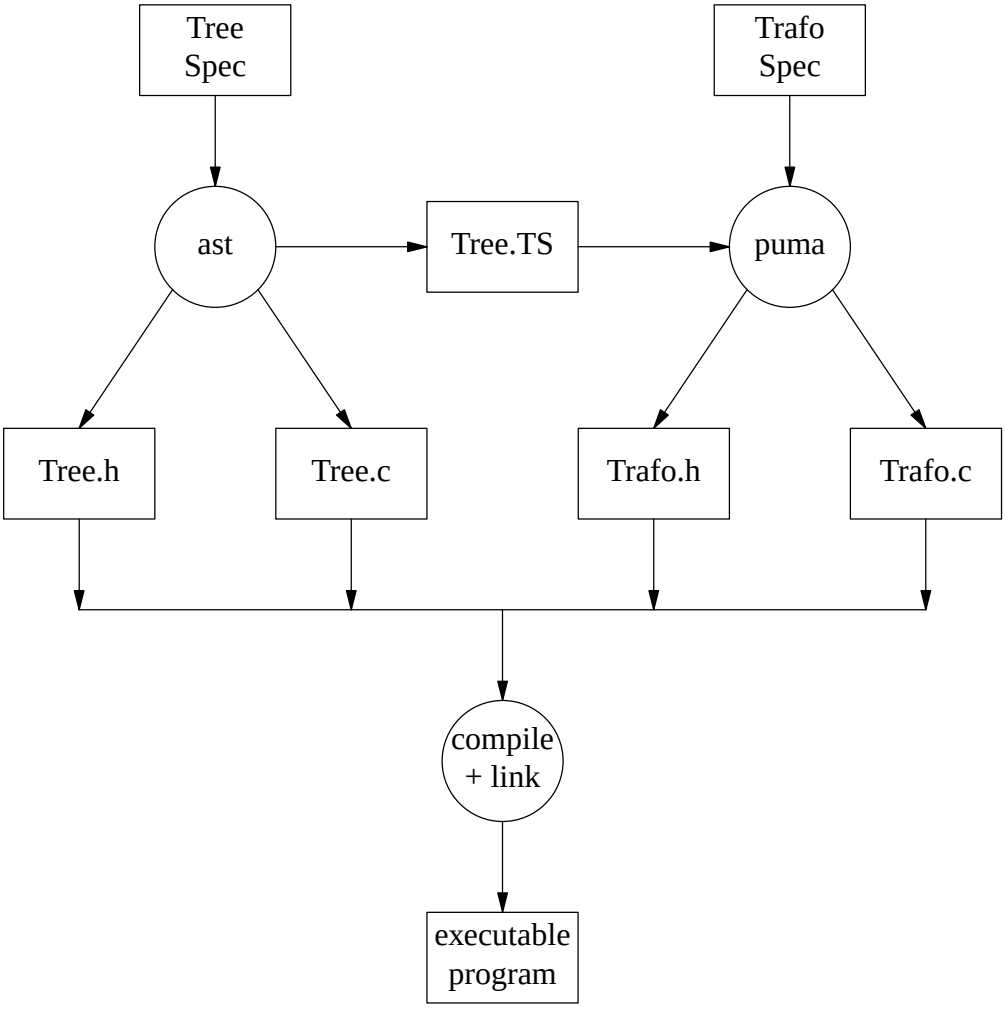

Fig. 1: Cooperation of *puma* and *ast*

attributes of the node types including their data types. This input language is described in the *ast* user manual [Groa].

3. Input Language

The following sections define the syntax and the semantics of a *puma* specification. Appendix 1 contains a summary of the precise syntax of the input language in BNF notation.

3.1. Notation

An EBNF notation is used in the following to describe the syntax of a *puma* specification. The meaning of the meta symbols is as follows:

=	introduces the right-hand side of a grammar rule
|	introduces alternatives (usually used to separate alternatives)
[ ]	square brackets enclose optional parts
{ }	curly brackets denote repetition zero, one, or more times
non alpha-numeric characters	terminal symbol
’ character ’	terminal symbol
all upper-case word	terminal symbol
other word	nonterminal symbol

3.2. Lexical Conventions

The input of *puma* consists of identifiers, numbers, keywords, operators, delimiters, comments, white space, and so called target code.

Identifiers are sequences of letters, digits, and underscore characters _ that start with a letter or an underscore character _. The case of the letters is significant. The single character _ is not treated as an identifier but as a don’t care symbol (see below):

```
x   NoName   k2   mouse_button
```

Numbers comprise integers and reals in decimal notation. They are written as in the target language:

```
0   007   1991   31.4E-1
```

The following words are reserved as keywords and may not be used as identifiers:

```
AND          BEGIN        CLOSE        CONDITION    DIV
DO           ELSE         ELSIF        END          EXPORT
EXTERN       FAIL         FOR          FUNCTION     GLOBAL
GUARD        IF           IMPORT       IN           LOCAL
MOD          NIL          NL           NOT          OR
PREDICATE    PROCEDURE    PUBLIC       REF          REJECT
RETURN       THEN         TRAFO        TREE         TRIP
WHILE
```

Operators are either symbols from the following list or sequences of characters introduced by a backslash \ and terminated by white space. Escaped operators are used for operators not known to *puma*. They are written to the output with the backslash \ removed.

```
!      !=     #      %      &      &&     *      +      ++     -      --
->     .      /      <      <<     <=     <>     =      ==     >      >=
>>     ^      |      ||     AND    DIV    IN     MOD    NOT    OR     ?  :
```

Examples of escaped operators:

```
\,    \?    \:    \(void)    \(int*)    \(struct \node)
```

The following symbols denote assignment symbols and assignment operators:

```
:=    =    +=    -=    *=    /=    %=    <<=    >>=    &=    ^=    |=
```

The following characters are delimiters:

```
(   )   ,   .   ..   ...   :   <:   :>   :-   ;   =>   [   ]   _   {   }   ?=
```

The delimiters `...` and `..` can be used alternatively. Comments are characters enclosed in /* and */ as in C which may not be nested or characters following // until the end of line as in C++:

```
/* comment */
// comment
```

Target code are declarations, statements, types, or expressions written in the target language and enclosed in curly brackets { }. Target code may contain curly brackets { } as long as these are either properly nested or contained in strings or in character constants. Unnested curly brackets outside of strings or character constants have to be escaped by a backslash character \. In general all characters outside of strings or character constants may be escaped by a backslash character \. This escape mechanism is not necessary in strings and character constants. Target code is usually copied unchecked and unchanged to the output.

```
{ x = 1; }
{ char * }
{ { char c = '}'; } }
{ printf ("}\n"); }
```

White space characters like blanks, tab characters, form feeds, and return characters are ignored.

3.3. Structure

The input of *puma* consists of several clauses, global declarations, target code sections, and a list of subroutines.

Syntax:

```
Input  = { Clause } { Subroutine }

Clause =
| TRAFO Ident
| TREE Idents
| PUBLIC Idents
| IMPORT String [;]
| EXTERN Idents [;]
| GLOBAL Declarations [;]
| TargetCodes

Idents = Ident { , Ident }
```

The identifier behind the keyword TRAFO determines the name of the generated module. The default name is *Trafo*.

The identifiers behind the keyword TREE refer to the tree modules to be manipulated. A *puma* module can not only handle one tree definition but an arbitrary number. There must be a tree grammar for every tree and they all must have been converted to their internal format with *ast*. More precisely, those names refer to so-called *views* of a tree definition. Roughly speaking, a view selects a subset of a tree definition. See the documentaion of *ast* for a description of this concept. If the

keyword TREE is missing then the following serves as default:

```
TREE Tree
```

Therefore an empty list of tree definitions has to be given as:

```
TREE
```

The identifiers behind the keyword PUBLIC specify those subroutines that should become visible from outside the module. External declarations for these subroutines are inserted automatically in the interface part of the generated module.

The string behind the keyword IMPORT specifies the filename of an other *puma* specification. The subroutines visible from outside of this specification become known to the current specification. This allows for complete type checking of calls of these external subroutines. The IMPORT directive can be repeated.

The identifiers behind the keyword EXTERN specify those identifiers of global, local, or external variables and subroutines that are used in some subroutines but that are not declared from the point of view of *puma*. They may be used in expressions and statements that are checked by the tool without causing a message.

The declarations behind the keyword GLOBAL, which are not enclosed in curly brackets, allow the declaration of global variables having as type a node type. For these variables *puma* does perform type checking.

Example:

```
TRAFO ICode TREE Tree Defs PUBLIC Code
IMPORT "Types.pum"
EXTERN ADD CHK ENT Emit
GLOBAL e: Expr, s: Stmts
```

3.4. Target Code

A *puma* specification may contain several sections containing *target code*. Target code is source code written in the target language. It is copied unchecked and unchanged to certain places in the generated module.

Syntax:

```
TargetCodes =
| EXPORT TargetCode
| GLOBAL TargetCode
| BEGIN  TargetCode
| CLOSE  TargetCode
```

The meaning of the different sections is as follows:

EXPORT:	declarations to be included in the interface part.
GLOBAL:	declarations to be included in the implementation part at global level.
BEGIN:	statements to initialize the declared data structures.
CLOSE:	statements to finalize the declared data structures.

Example in C or C++:

```
EXPORT { typedef int MyType; extern MyType Sum; }
GLOBAL {# include "Idents.h"
          MyType Sum; }
BEGIN  { Sum = 0; }
CLOSE  { printf ("%d", Sum); }
```

Example in Java:

```
EXPORT { public int member; }
GLOBAL { # define beginMyType(a) a = 0; }
BEGIN  { ... }
CLOSE  { ... }
```

Example in Modula-2:

```
EXPORT { TYPE MyType = INTEGER; VAR Sum: MyType; }
GLOBAL { FROM Idents IMPORT tIdent; }
BEGIN  { Sum := 0; }
CLOSE  { WriteI (Sum, 0); }
```

3.5. Subroutines

A set of subroutines constitutes the main building blocks of a transformation. Like in programming languages, subroutines are parameterized abstractions of statements or expressions. There are four kinds of subroutines:

procedure	: a subroutine acting as a statement
function	: a subroutine acting as an expression and returning a value
predicate	: a boolean function
trip	: a procedure with automatic tree traversal

Syntax:

```
Subroutine = Header [ EXTERN Idents ; ] [ LOCAL TargetCode ]
             [ LOCAL Declarations ; ] { Rule }

Header     =
| [ PUBLIC ] PROCEDURE Ident ( [ Parameters ] [ => Parameters ] )
| [ PUBLIC ] FUNCTION  Ident ( [ Parameters ] [ => Parameters ] ) Type
| [ PUBLIC ] PREDICATE Ident ( [ Parameters ] [ => Parameters ] )
| [ PUBLIC ] TRIP      Ident ( [ Parameters ]                   )

Parameters = [ REF ] [ Ident : ] Type { , [ REF ] [ Ident : ] Type }
```

A subroutine consists of a header, a list of external identifiers, a target code section, local declarations and a sequence of rules. Except for the header all components are optional. The header specifies the kind of the subroutine, its name, and its parameters. In case of a function, the type of the result value is added. This type is restricted to node types and types which are legal for function results in the target language (usually simple types and pointers). Input and output parameters are separated by the symbol =>. It suffices to give the type of a parameter. A name for the formal parameter is optional. Usually input parameters are passed by value and output parameters are passed by reference. The keyword REF can be used to pass input parameters by reference, too. This might be necessary in case of tree modifications when an input tree is replaced by a newly created one.

The keyword PUBLIC specifies that the subroutine should become visible from outside the compilation unit. This is equivalent to including the name of the subroutine into the list of identifiers after the keyword PUBLIC at the beginning of a specification. The identifiers behind the keyword EXTERN specify those identifiers of global, local, or external variables and subroutines that are used within the subroutine but that are not declared from the point of view of *puma*. They may be used in expressions and statements that are checked by the tool without causing a message. The target code section after the keyword LOCAL is copied in front of the body of the generated subprogram and may e. g. contain local declarations. The declarations behind the keyword LOCAL, which are not enclosed in curly brackets, allow the declaration of local variables having as type a node type. For these variables *puma* does perform type checking.

Examples:

```
PROCEDURE Code (t: Tree) LOCAL { tObjects object; } LOCAL e: Expr ...
PREDICATE IsCompatible (Type, Type) ...
FUNCTION  ResultType (Type, Type, int) Type ...
PROCEDURE ResultType (Type, Type, int => Type) ...
TRIP doit ([Expr, Stmts], Tree, int) ...
```

3.5.1. Trip

A subroutine of kind *trip* is a procedure with automatic tree traversal. It has to have at least one parameter. It can have any number of input parameters, output parameters are not possible. The semantics of output parameters can be achieved by passing input parameters by reference using the keyword REF. The first parameter must be a node type. It specifies the tree to be traversed automatically. Unlike the procedures *TraverseTreeTD* and *TraverseTreeBU* generated by the tool *ast* a trip can automatially traverse trees, only. It can not traverse graphs - except some kind of marking algorithm is added manually.

The exact definition of a trip is as follows. If a trip t has a header like this one:

```
TRIP t (S, p1: T1, ..., pn: Tn)
```

Then the set of rules given for t is extended by appending a rule such as the following for every node type N in the set of node types S:

```
N (c1 := c1, ..., cm := cm) :-
   t (c1, p1, ..., pn);
   ...
   t (cm, p1, ..., pn);
```

This describes a traversal of the tree using recursive calls of the trip t for the children c1 through cm. Recursive calls are generated only for those children, whose type is in the set of node types S. The sequence of the visits of the children is from left to right, except for children having the property REVERSE, which are visited last. The parameters p1 through pn are passed unchanged by the recursive calls.

The region to be traversed automatically can be described by the set of node types S:

Examples:

```
TRIP t1 (Stmts)
TRIP t2 ([Stmts, Expr])
TRIP t3 (Tree)
```

The trip t1 would traverse nodes of type *Stmts*, only. The trip t2 would traverse nodes of type *Stmts* and *Expr*. The trip t3 would traverse all node types of the tree.

The automatic traversal can be changed by giving explicit instructions. In the following examples the function call A stands for an arbitrary action statement.

Examples:

```
TRIP t (Expr)

Unary  (...) :- .                                  /* do not visit subtree   */
Binary (Lop, Rop) :- A (); t (Rop); t (Lop); .     /* change visit sequence  */
Binary (Lop, Rop) :- t (Lop); t (Rop); A (); .     /* bottum-up traversal    */
Binary (Lop, Rop) :- A (); t (Lop); t (Rop); .     /* top-down traversal     */
Binary (Lop, Rop) :- A (); REJECT .                /* fall back to automatic */
```

Note, the last rule implements top-down traversal by using the rules added for automatic traversal. All node types in the class *Expr* other than *Unary* and *Binary* are traversed automatically according to the default visit sequence.

3.6. Types

Types are either predefined in the target language like *int* and *INTEGER*, or user-defined like *MyType*, or they are tree types like *Expr*. A tree type is described by the name of a tree definition, a single node type, or a list of node types enclosed in brackets [ ]. In case of ambiguities the latter two kinds may be qualified by preceding the name of the tree definition. In every case a tree-type defines a set of legal node types. The name of a tree definition refers to every node type that is defined there. A single node type yields a set with just this one element and a list of node types yields the union of all list elements.

Syntax:

```
Type     =
| TreeType
| UserType

TreeType =
| Ident
| [ Ident . ] Ident
| [ Ident . ] '[' Idents ']'

UserType =
| Ident
| TargetCode
```

Examples:

```
int                     /* predefined type            */
MyType                  /* user defined type          */
{ char * }              /* type expression            */
Tree                    /* tree type                  */
Expr                    /* node type                  */
Tree.Expr               /* qualified node type        */
[Stats, Expr]           /* set of node types          */
Tree.[Stats, Expr]      /* qualified set of node types */
```

3.7. Rules

A rule behaves like a branch in a case or switch statement. It consists of a list of patterns (nonterminal Patterns), a list of expressions, a return expression in case of a function, and a list of statements. Several neighbouring rules with the same list of expressions, return expression, and list of statements may share those parts. A list of a list of patterns (nonterminal PatternList) is equivalent to a

sequence of rules having the sublists as patterns and sharing the other parts.

Patterns and expressions may be either positional or named. Positional entities consist just of a pattern or an expression. They have to be written in the same order as the corresponding parameters. Named entities consist of a selector name and a pattern or an expression. The selector name specifies a child or an attribute. The named entities can be written in any order and it is not necessary to give an entity for every parameter. The named entities have to follow the positional ones. For every position of a pattern or an expression at most one entity may be given. The named elements are transformed into their positional form before type checking is performed. It is recommended to use only named entities because this way the rules remain valid while the abstract syntax is changing.

The parts of a rule may be given in almost any order as described by the exact syntax in Appendix 1.

The number of patterns must agree with the number of input parameters, and the types of the elements of those lists must be pairwise compatible. The number of expressions must agree with the number of output parameters, and the types of the elements of those lists must be pairwise compatible. The type of the expression after RETURN has to be compatible with the result type of a function. The type s of a pattern or an expression is said to be compatible to the type t of a formal parameter if s is a subtype of t ($s \subseteq t$).

Syntax:

```
Rule         = [ PatternList ] [ => Exprs ] [ RETURN Expr ] :- { Statement ; } .

PatternList  = Patterns { ; Patterns }

Patterns     =
| Pattern { , Pattern } { , NamedPattern }
| NamedPattern          { , NamedPattern }

NamedPattern = SelectorName := Pattern

Exprs        =
| Expr { , Expr } { , NamedExpr }
| NamedExpr       { , NamedExpr }

NamedExpr    = SelectorName := Expr

SelectorName = Ident
```

The semantics of a rule is as follows: A rule may succeed or fail. It succeeds if all its patterns, statements, and expressions succeed - otherwise it fails. The patterns, statements, and expressions are checked for success in the following order: First, the patterns are checked from left to right. A pattern succeeds if it matches its corresponding input parameter as described below. Second, the statements are executed in sequence as long as they succeed. The success of statements is defined below. Third, the expressions are evaluated from left to right and their results are passed to the corresponding output parameters. In case of a function, additionally the expression after RETURN is evaluated and its result is returned as value of the function call. The success of expressions is defined below, too. If all elements of a rule succeed then the rule succeeds and the subroutine returns. If one element of a rule fails the process described above stops and causes the rule to fail. Then the next rule is tried. This search process continues until either a successful rule is found or the end of the list of rules is reached. In the latter case the behaviour depends on the kind of the subroutine:

A procedure signals a runtime error if option 'f' is set, otherwise it does nothing.
A predicate returns false.
A function signals a runtime error.
A trip executes its default rules, if any, otherwise it does nothing.

There is one exception to this definition of the semantics which is explained later. Note, if a predicate fails then the values of its output parameters are undefined.

Examples:

```
PROCEDURE Code (t: Tree)
   Plus  (Lop, Rop) :- Code (Lop); Code (Rop); Emit (ADD); .
   Minus (Lop, Rop) :- Code (Lop); Code (Rop); Emit (SUB); .
   ...

PREDICATE IsCompatible (Type, Type)
   Integer (), Integer () :- .
   Real    (), Real    () :- .
   Boolean (), Boolean () :- .
   Array (t1, Lwb, Upb, _), Array (t2, Lwb, Upb, _) :- IsCompatible (t1, t2); .

FUNCTION ResultType (Type, Type, int) Type
   Integer (), Integer (), (Plus )  RETURN Integer () :- .
   Real    (), Real    (), (Plus )  RETURN Real    () :- .
   Integer (), Integer (), (Times)  RETURN Integer () :- .
   Real    (), Real    (), (Times)  RETURN Real    () :- .
   Integer (), Integer (), (Less )  RETURN Boolean () :- .
   Real    (), Real    (), (Less )  RETURN Boolean () :- .
```

3.8. Patterns

A pattern describes the shape at the top or root of a subtree. A pattern can be a decomposition of a tree, the keyword NIL, a label or a variable, one of the don't care symbols _ or ..., or an expression. A decomposition is written as a node type followed by a list of patterns in parenthesis (and). Optionally, the node type may be qualified by a tree name and preceded by a label.

Syntax:

```
Pattern =
| [ Label ] [ Ident . ] Ident [ ( [ Patterns ] ) ]
| [ Label ] NIL
| Ident
| _
| ...
| Expr

Label   =
| Ident :
| Ident :>
```

The match between a pattern and a value is defined recursively depending on the kind of the pattern:

- A decomposition with a node type t matches a tree u with a root node of type s if u is not equal to NoTree (NULL, null, NIL) and s is a subtype of t ($s \subseteq t$) and all subpatterns of t match their corresponding subtrees or attributes of u. If the node type is preceded by a label l then a binding is established between l and u which defines the label l to refer to the tree u. If the label l is followed by a colon : then l has the type of u. If the label l is followed by the symbol :> then l has the type that is legal at this location. This is either the type of a parameter or the type of a

node type's child.

- The pattern NIL matches the value NoTree (NULL, null, NIL). If NIL is preceded by a label l then a binding is established between l and the parameter or child matching NIL. l has the type that is legal at this location. This is either the type of a parameter or the type of a node type's child.
- The first occurrence of a label l in a rule matches an arbitrary subtree or attribute value u. A binding is established between l and u which defines the label l to refer to the value u. The label can be used later to access the associated value. All further occurrences of the label l within patterns of this rule match a subtree or an attribute value v only if u is equal to v. The equality for trees is defined in the sense of structural equivalence. Two attributes are equal if they have the same values. This so-called non-linear pattern matching has to be enabled by an option. Without this option all further occurrences of a label l are treated as error.
- The don't care symbol _ matches one arbitrary subtree or attribute value.
- The don't care symbol . . . matches any number of arbitrary subtrees or attribute values.
- An expression matches a parameter or an attribute if both have the same value. The equality of values is defined as a type specific operation (see section 3.11.).

By default, all node types have to be followed by parentheses, otherwise they are considered as labels. The option -p of *Puma* allows parentheses after node types to be omitted. A node type t without a list of subpatterns enclosed in parentheses is treated as t (. . .). The ambiguity between a node type without a list of patterns in parentheses and a label is resolved in favor of the node type.

Examples:

```
Binary (Lop, Rop, Operator)   /* a node type                          */
Binary                        /* a node type (requires option -p)     */
Tree.Binary                   /* a qualified node type (requires -p)  */
a:Binary (_, b:>Binary (Lop, ...), Operator)
                              /* a, b, Lop, and Operator are labels   */
                              /* a is of type Binary                  */
                              /* b is of type Expr                    */
NIL                           /* NULL, null, or NoTree                */
X                             /* a label                              */
k + 2                         /* an expression                        */
(Times)                       /* a named constant                     */
{Times}                       /* a named constant                     */

Binary (Lop, Rop, ...)        /* positional pattern                   */
Binary (Lop, Rop := Rop)      /* named patterns                       */
Binary (Lop := Lop, Rop := Rop)
Binary (Rop := Rop, Lop := Lop)
Binary (Lop := Lop: Lop, Rop := Rop:> Rop)
Binary (Lop := Lop: Binary (Lop, ...), Rop := Rop:> Rop)
```

3.9. Expressions

Expressions denote the computation of values or the construction of trees. Binary and unary operations as well as calls of external functions are written as in the target language. Calls of *puma* functions and predicates distinguish between input and output arguments. Named arguments are not allowed in calls. The syntax for tree composition is similar to the syntax of patterns. Again, the node type may be qualified by a tree name.

Syntax:

```
Expr =
| [ Ident . ] Ident [ ( [ Exprs ] ) ]
| NIL
| Ident
| _
| ...
| Expr ( [ Exprs ] [ => Patterns ] )
| GUARD Expr
| <: Expr
| Expr Operator Expr
| Operator Expr
| Expr Operator
| Expr '[' Exprs ']'
| ( Expr )
| Expr ? Expr : Expr
| ( Type ) Expr
| Number
| { String }
| TargetCode
| Ident :: Ident
```

The semantics of the different kinds of expressions is as follows:

- A name of a node type followed by parentheses creates a tree node and provides the children and attributes of this node with the values given in parenthesis. If option -p is given then a missing list in parentheses is treated as (. . .).
- NIL represents the value NoTree (NULL, null, NIL).
- A label refers to the expression it was bound to upon its definition.
- A function or predicate call must be compatible with the corresponding definition in terms of the numbers of expressions and patterns as well as their types. A function call evaluates the expressions corresponding to input parameters, passes the results to the function, and executes the function. Upon return from the function the result value of the function determines the result of this expression. The values of the output parameters that the function returns are matched against the actual patterns of the function call. If one pair does not match the call fails. Labels in the patterns may establish bindings that enable to refer to the output parameters or subtrees thereof.
- The don't care symbols specify that no computation should be executed, either for one or for several expressions. The result values are undefined.
- The keyword GUARD or the symbol <: in front of an expression denote a so-called type-guard. Usually, *puma* does type checking during generation time. Using a type-guard type checking is performed during runtime. At runtime the type of the expression is checked whether it is a subtype of the type required by the context of the expression. If the subtype condition does not hold then the expression fails and the execution of the current rule is terminated.
- The most common binary and unary operators (prefix and postfix) of the target language as well as array indexing and parentheses are known to *puma*. This is true as well for the conditional expression and simple type casts of C. They are passed unchanged to the output.
- A target code expression, a number, or a string is evaluated as in the target language.

- The construct Ident :: Ident can be used to refer to children or attributes that are not matched by a label. This can be of interest because of notational brevity or because matching is impossible. The reason for the latter case can arise when a subset of a tree definition is presented to *puma* using the concept of views. The first identifier is a label that is bound to a tree (node). The second identifier is the name of a child or of an attribute of this node type.

In case of node types, labels for tree values, and functions returning tree values, *puma* does type checking. For user types, target code expressions or target operators no type checking is done by *puma* but (hopefully) later by the compiler. An expression that does not contain calls of *puma* functions or predicates always succeeds. An expression containing those calls succeeds if all the calls succeed – otherwise it fails.

Examples:

```
Binary (X, Y, Z)          /* a node composition                        */
Binary                    /* a node composition (requires option -p)   */
Tree.Binary               /* a node composition (requires option -p)   */
NIL                       /* NULL, null, or NoTree                     */
X                         /* a label or a variable                     */
ResultType (t1, t2)       /* a function call                           */
_                         /* don't care (one expression)               */
GUARD X                   /* a type-guard                              */
<: X                      /* a type-guard                              */
k + 2                     /* an expression                             */
- k                       /* an expression                             */
k ++                      /* an expression                             */
a [x]                     /* an expression                             */
a == b ? c : d            /* a conditional expression                  */
(char *) p                /* a type cast                               */
(Times)                   /* a named constant                          */
{Times}                   /* a named constant                          */
3.14                      /* a constant                                */
"abc"                     /* a constant                                */
```

3.10. Statements

Statements are used to describe conditions, to perform output, to assign values to attributes, and to control the execution of the transformer using branching, repetition, and recursive subroutine calls. A statement is either a condition denoted by an expression optionally preceded by the keyword CONDITION, a call of a procedure, an assignment, a control flow statement starting with one of the keywords IF, FOR, WHILE, RETURN, REJECT, or FAIL, a String or the keyword NL, a target code statement, or a declaration of variables. Named arguments are not allowed in procedure calls. Every kind of statement may succeed or fail as described below.

Syntax:

```
Statement    =
| Expr
| CONDITION Expr
| Expr ( [ Exprs ] [ => Patterns ] )
| Expr AssignSymbol Expr
| Expr AssignOperator Expr
| Expr ?= Pattern
| CONDITION Expr ?= Pattern
| IF Expr THEN { Statement ; } [ Elsifs ] END
| IF MatchConditions THEN { Statement ; } [ Elsifs ] END
| FOR [ InitStmt ] ; Expr ; [ IncrStmt ] ; DO { Statement ; } END
| WHILE Expr DO { Statement ; } END .
| FOR Ident [ : Type ] IN Expr DO { Statement ; } END
| RETURN Expr
| RETURN
| REJECT
| FAIL
| { String }
| NL
| TargetCode
| Declarations

AssignSymbol =
| =
| :=

Elsifs       =
| ELSIF Expr THEN { Statement ; } [ Elsifs ]
| ELSIF MatchConditions THEN { Statement ; } [ Elsifs ]
| ELSE { Statement ; }

MatchConditions =
             [ Expr && ] Expr ?= Pattern { ; [ Expr && ] Expr ?= Pattern }

InitStmt     =
| IncrStmt
| Ident : Type AssignSymbol Expr

IncrStmt     =
| Expr AssignSymbol Expr
| Expr AssignOperator Expr

Declarations = Declaration { , Declaration }

Declaration  = Ident : Type [ AssignSymbol Expr ]
```

There are some syntactic ambiguities: Target code in curly brackets { } is considered as target code statement instead of as target code expression. To obtain the latter meaning the expression should be enclosed in parentheses ( ). Subroutine calls are treated according to their declaration: Predicates and functions are treated as conditions, procedures and external subroutines are treated as procedure calls. If external subroutines should be considered as expressions, the call should be enclosed in parentheses ( ), too. A string is considered as a special kind of statement instead of as a normal expression.

- Conditions are denoted by expressions optionally preceded by the keyword CONDITION and can be used to determine properties that can not be expressed with pattern matching alone. Patterns describe either shapes of a fixed size of a tree or the equality between two values. Properties of trees of unlimited size and relations like <, <= etc. have to be checked with conditions.

The expression has to be of type boolean or the call of a predicate. A condition succeeds if the expression evaluates to true - otherwise it fails.

- For a procedure call the same rules as for a function call apply. It succeeds if the values of all output parameters are matched by the corresponding patterns - otherwise it fails. A call of an undefined subroutine is treated as a call of a procedure that is either defined externally or in the GLOBAL target code section. Such a call is flagged by a warning message.
- An assignment statement evaluates its right-hand side expression and stores the resulting value at the entity denoted by the identifier on the left-hand side. The identifier can denote

  a global or a local variable,
  an input or an output parameter, or
  a label for an attribute or a subtree.

  An assignment statement succeeds if the expression succeeds - otherwise it fails.
- A so-called match statement consists of an expression and a pattern optionally preceded by the keyword CONDITION:

  ```
  [ CONDITION ] Expr ?= Pattern ;
  ```

  The statement behaves like a condition. The expression is matched against the pattern. If the match fails then the statement fails and the execution of the current rule is terminated. Otherwise, the variables in the pattern are bound to the values they match and the execution of the current rule continues with the next statement.
- The IF-THEN-ELSE statement can be used to describe control flow for branching. This statement succeeds if all contained statements and expressions succeed - otherwise it fails. The condition can be a usual boolean expression or a list of so-called match conditions separated by semicolon characters. A match condition has the following form:

  ```
  Expr ?= Pattern
  ```

  The expression is matched against the pattern. If the match succeeds then the condition is regarded to be true. Additionally, all pattern variables in the pattern are bound. They are visible and can be used in the THEN part, only. A list of match conditions such as

  ```
  IF Expr1 ?= Pattern1 ; Expr2 ?= Pattern2 THEN Stmts END
  ```

  is equivalent to the following:

  ```
  IF    Expr1 ?= Pattern1 THEN Stmts
  ELSIF Expr2 ?= Pattern2 THEN Stmts
  END
  ```

  Every match condition can optionally be combined with a boolean condition using the operators && or AND:

  ```
  IF Cond && Expr ?= Pattern THEN Stmts END
  IF Cond1 && Expr1 ?= Pattern1 ; Cond2 && Expr2 ?= Pattern2 THEN ...
  ```

  The complete condition evaluates to true only if the boolean condition yields true and if the match condition succeeds.
- The FOR and WHILE statements can be used to describe control flow for repetition similar to the *for* and *while* statements in C. The FOR statement allows the declaration of a loop variable similar to C++. These statements succeed if all contained statements and expressions succeed - otherwise they fail.
- The FOR IN statement provides iteration over the elements of a list of tree nodes. Lists of tree nodes are defined by the property REVERSE. The child with this property refers to the next

element in the list. The iteration variable can be declared using the optional type clause.

- The RETURN statement can be used in functions and procedures in order to terminate the subroutine and to return the value of the expression after RETURN.
- The statement REJECT does nothing but fail. This way the execution of the current rule terminates and control is passed to the next rule.
- The statement FAIL causes the execution of the current subroutine to terminate. This statement is allowed in procedures and predicates, only. Depending on the kind of subroutine the following happens:

  A procedure and a trip terminate.
  A predicate returns false.

- A string is an output statement that prints this string. (For details see section 3.13.).
- The keyword NL is an output statement that prints a newline character. (For details see section 3.13.).
- A target code statement is executed as in the target language. It can be used for arbitrary actions. In particular it can compute the value of an explicitly declared label (variable) by means of implementation language code or calls of external subroutines. A target code statement always succeeds.
- A declaration explicitly introduces a label or variable in the scope of a rule. This entity can optionally be initialized with a value given by an expression. Without initialization its value is undefined. It can be used for the definition of temporary variables. The user is responsible that these variables receive values either by initialization, by assignments, or by target code statements. A declaration without initialization always succeeds. A declaration with initialization succeeds if the expression succeeds - otherwise it fails.

Note, statements and expressions may cause side effects by changing e. g. global variables, local variables, the input tree, or by producing output. Those side effects are not undone when a rule fails.

Examples:

```
CONDITION X < Y               /* condition                              */
IsCompatible (t1, t2)         /* condition: predicate call              */
(IsSimpleType (t))            /* condition: external  call              */
X < Y                         /* condition: expression                  */
({ X < Y })                   /* condition: target code expression      */
Code (Then)                   /* procedure call: internal               */
printf ("hello")              /* procedure call: external               */
X = Y                         /* assignment in C, condition in Modula-2 */
X := Y                        /* assignment in both C and Modula-2      */
X += Y                        /* assignment operator                    */
{ X = Y; }                    /* assignment using target code           */
X ++                          /* statement (not a condition)            */
X ?= Binary (Lop, ...)        /* match statement                        */
IF Z THEN X = Y; END
IF Z THEN X = Y; ELSE Y = X; END
IF Z THEN X = Y; ELSIF ZZ THEN Y = X; END
IF Z THEN X = Y; ELSIF ZZ THEN Y = X; ELSE Z = 0; END
IF X ?= Binary (Lop := L) THEN Y = X; END   /* match condition          */ |
IF X ?= Binary (Lop := L); Y ?= Unary  (...) THEN Y = X; END               |
FOR i: int = 1; i <= 10; i ++ DO printf ("hello"); END                     |
WHILE i <= 10 DO printf ("hello"); i ++; END
FOR s IN Stats DO process (s); END                                        |
FOR s: Stat IN Stats DO process (s); END                                  |
REJECT                        /* terminate rule, try next rule          */
FAIL                          /* terminate subroutine and return        */
"hello"                       /* print string                           */
NL                            /* print newline character                */
{ Code (Then); }              /* unchecked internal call                */
{ printf ("hello"); }         /* unchecked external call                */
Z: Expr                       /* declaration without initialization     */
Z: Expr = Binary (x, Y, ...)  /* declaration with initialization        */
{ Z = mBinary (X, Y, Plus); } /* assignment using target code           */
```

3.11. Equality Operations

The equality between two trees is defined recursively: Two trees are equal if the node types of the two root nodes are equal and all corresponding subtrees or attributes are equal.

The equality between attribute values is type specific. For every type name a separate equality test is defined. Chosing different type names for one type introduces subtypes and allows to treat attributes of different subtypes differently. The equality tests are defined by a macro mechanism using the C preprocessor *cpp*:

```
# define equalTYPE(a, b) (a) == (b)
```

TYPE is replaced by the concrete type name. *a* and *b* are formal macro parameters referring to the attributes to be compared.

The equality test for the predefined types of a target language are predefined within *puma* (see Appendix 3). For user-defined types, by default the following equality test is used:

in C or C++:

```
# define equalTYPE(a, b) memcmp ((char *) & a, (char *) & b, sizeof (a)) == 0
```

in Java:

```
# define equalTYPE(a, b) (a.equals (b))
```

in Modula-2:

```
# define equalTYPE(a, b) yyIsEqual (a, b)
```

Above procedures check values of arbitrary types by comparing the byte sequences.

It is possible to redefine the operations by including new macro definitions in the GLOBAL section. The following example demonstrates the syntax for doing this.

Example in C or C++:

```
GLOBAL {
typedef struct { short Line, Column; } tPosition;
# define equaltPosition(a, b) a.Line == b.Line && a.Column == b.Column
}
```

Example in Java:

```
GLOBAL {
# define equaltPosition(a, b) (a.compareTo (b) == 0)
}
```

Example in Modula-2:

```
GLOBAL {
TYPE tPosition = RECORD Line, Column: SHORTCARD; END;
# define equaltPosition(a, b) (a.Line = b.Line) AND (a.Column = b.Column)
}
```

3.12. Begin Operations

Usually, a composition of a node specifies values for the attributes and children. Using dont't care symbols it is possible to omit these values. In this case the attributes and children are initialized either by an initializer expression, if specified, or a macro mechanism using the C preprocessor *cpp*:

```
# define beginTYPE(a)
```

TYPE is replaced by the concrete type name. *a* is the formal macro parameter referring to the attribute or children to be initialized.

Initialization for attributes without initializer expression is done within *puma* by macros which are predefined to be empty for most of the types. Exceptions are the types *tStringRef*, *tIdent*, and *tPosition* which are initialized with the values *NoString*, *NoIdent*, and *NoPosition*. Children are set to NoTree (NULL, null, NIL) by default:

in C or C++:

```
# define beginTYPE(a) a = NULL;
```

in Java:

```
# define beginTYPE(a) a = null;
```

in Modula-2:

```
# define beginTYPE(a) a := NIL;
```

It is possible to redefine the operations by including new macro definitions in the GLOBAL section. The following example demonstrates the syntax for doing this.

Example in C, C++, or Java:

```
GLOBAL {# define beginint(a) a = 0;}
```

Example in Modula-2:

```
GLOBAL {# define beginINTEGER(a) a := 0;}
```

3.13. Output Statements

The two builtin output statements "string" and NL are translated into macro calls:

```
yyWrite ("string");
yyWriteNl;
```

The macros are predefined as follows:

in C or C++:

```
# define yyWrite(s) yyWriteString (s)
# define yyWriteNl  yyWriteString ("\n")

static FILE * yyf = NULL;

static void yyWriteString (const char * yyString)
{
 if (yyf == NULL) yyf = stdout;
 fputs (yyString, yyf);
}
```

in Java:

```
# define yyWrite(s) yyf.write (s)
# define yyWriteNl  yyf.writeNl ()
```

in Modula-2:

```
# define yyWrite(s) IO.WriteS (yyf, s)
# define yyWriteNl  IO.WriteNl (yyf)

VAR yyf: IO.tFile;

yyf := IO.StdOutput;
```

By default the statements print on standard output using the library routines specified in the macro definitions. This behaviour can be changed in two ways: The global variable yyf can be assigned a new value that describes an arbitrary output file or the macros can be redefined in the GLOBAL target code section. For example in C or C++ a more efficient version of the macros is as follows, provided that the variable yyf has a defined value:

```
# define yyWrite(s) fputs (s, yyf)
# define yyWriteNl  fputc ('\n', yyf)
```

4. Scopes

Scopes are regions of the specification text which control the meaning of identifiers. An identifier with a certain meaning can be used in its scope. The same identifier may have a different meaning in a different scope. The same identifier may have different meanings within one scope as well, where the meaning depends on the context. A *puma* specification distinguishes the following kinds of scopes which are nested in each other:

External Scope
: The external scope contains all subroutines from other *puma* specifications which are imported with the IMPORT directive.

Global Scope
: A complete *puma* specification defines a global scope. It contains all declarations included in the GLOBAL target code section, all GLOBAL declarations, and all subroutine definitions. The subroutines can be defined in any order.

Subroutine Scope
: Every subroutine definition introduces a local scope. It contains the names of the input and output parameters, the declarations included in a LOCAL target code section, and the LOCAL declarations.

Rule Scope
: Every rule introduces a rule scope. It contains the labels, pattern variables, and declared variables which are defined in this rule except for definitions in statement scopes (nested statements, see below). Labels and pattern variables are declared upon their first occurrence in patterns. They are visible only within a rule. Labels in expressions represent using positions. Labels have to be declared or bound textually before they are used.

Statement Scopes
: The statements which might have other statements as their constituents (nested statements) introduce statement scopes. The bodies of the FOR and WHILE loops as well as the THEN and the ELSE parts of an IF statement are statement scopes. These scopes contain the labels, pattern variables, and declared variables which are defined in these regions except for definitions in nested statement scopes. Moreover, a variable declared in the header of a FOR loop can be used in the header and in the body of this loop. If an IF statement contains a match condition then the labels or pattern variables defined by the match condition can be used in the THEN part of the IF statement, only.

Note, the global scope and the subroutine scopes can redefine the meaning of identifiers of enclosing scopes. This is not possible for rule scopes and statement scopes.

5. Output

From a given specification, *puma* generates a program module in one of the target languages C, C++, Java, or Modula-2 implementing the desired transformation. The subroutines in the sense of *puma* are mapped to subroutines in the target language. Procedures and trips yield procedures, functions yield functions that return a value, and predicates yield boolean functions. These subroutines can be called from other modules using the usual subroutine call syntax of the target language provided they are exported: All arguments are separated by commas - the symbol => as separator between input and output arguments is only required in calls processed by *puma*.

The types of the parameters are treated as follows: Predefined types or user defined types remain unchanged. Node types or sets of node types are replaced by the name of the corresponding tree type. This is a pointer to a union of record types. Input parameters are passed by value and output parameters are passed by reference (VAR in Modula-2) by default. Input parameters with the keyword REF are passed by reference, too.

In addition to the exported subroutines, a *puma* generated module exports the subroutines BeginTRAFO and CloseTRAFO, where TRAFO is replaced by the module name. Both subroutines contain the target code sections BEGIN and CLOSE. All target code sections and target code representing expressions or statements are more or less copied unchecked and unchanged to the generated output module. The only change is that in target code representing expressions or statements label identifiers are replaced by access paths to the associated values.

The rules of a subroutine are treated like a comfortable case or switch statement. The code generated for pattern matching is relatively simple. A naive implementation would just use a sequence of if statements. This kind of code showed to be already rather efficient. Possible optimizations are the clever use of switch statements and the elimination of common subexpressions. Furthermore, tail recursion can be turned into iteration. Labels are replaced by access paths to the associated values. The code for the construction of tree nodes is inserted in-line. It is therefore efficient because no procedure calls are necessary for the creation of tree nodes. Moreover, the transformer module is independent of the tree module with respect to the presence of procedures to create nodes and the classification of input attributes.

5.1. C++

5.1.1. Simple C++

The program modules generated by *puma* do not use any C++ specific constructs. The generated code uses the C subset of C++ and thus can be translated with C++ compilers. Both, the tree module and the transformation module have to be generated in simple C++ using the options -+ or -c+. In some cases it is necessary for the transformation module to call methods of the tree object, for example for the creation of new tree nodes. As there might be several tree objects it is necessary to describe which one to refer to. This is handled by generating code like this:

```
TREE_PREFIX MakeTREE (k<node_type>)
```

In order to make this code work, two declarations have to be included in the GLOBAL section such as for example:

```
# define TREE_PREFIX t->
extern TREE * t;
```

5.1.2. Proper C++

The program modules generated by *puma* make use of C++ specific language constructs. Both, the tree module and the transformation module have to be generated in proper C++ using the options -c++.

5.1.3. Idents_PREFIX

In the case of C++ the module *Idents* of the library *reuse* [Grob, Groc] is implemented as a class. Thus several objects might be created from this class. If attributes of the underlying tree are declared with the type *tIdent* then *puma* will generate code that uses some methods of this class. Now it has to be described to which of the objects a method call refers to. This is handled by

generating code like this:

```
# ifndef Idents_PREFIX
# include "Global.h"
# define Idents_PREFIX gIdents.
# endif

Idents_PREFIX NoIdent
```

By default, the globally created object *gIdents* from the library *reuse* is used. This can be changed by providing own definitions for the macro *Idents_PREFIX* in the GLOBAL section. Example:

```
# include "Idents.h"

# define Idents_PREFIX my_idents_object.

Idents my_idents_object;
```

6. Usage

NAME

puma - a generator for the analysis and transformation of attributed trees

SYNOPSIS

puma [-options] [-y *type*] [-z *type*] [-I *directory*] [-l *directory*] [*file*]

DESCRIPTION

puma is a tool for the analysis and transformation of attributed trees based on pattern matching. It generates transformers (named *Trafo* by default) that map attributed trees to arbitrary output. As this tool also has to know about the structure of the tree this information is communicated from *ast* to *puma* via a file with the suffix .TS. If *file* is omitted the specification is read from standard input.

OPTIONS

a generate all, same as -di (default)

d generate header file or definition module

i generate implementation part or module

s suppress information and warning messages

u suppress information messages

m use procedure MakeTREE to construct nodes (default: in-line code)

p allow node constructors without parentheses

f signal a runtime error if none of the rules of a procedure matches

k allow non-linear patterns

n check parameters for NoTREE (NULL, null, NIL) and treat as failure

w surround actions by WITH statements

e treat undefined names as error

v treat undefined names as warning

o list undefined names on standard output

t print tree definitions

r print patterns

x generate and print a traversal procedure for all node types (Traverse)

y *type*
generate and print a traversal procedure for all node types that are reachable from the node type *type*

z *type*
restrict traversal procedure to reach node types of class *type*, only

q browse internal data structure with text browser

: generate lines not longer than 80 characters

6 generate # line directives

7 touch output files only if necessary

8 report storage consumption

c generate C source code (default: Modula-2)

c+ generate simple C++ source code

c++ generate proper C++ source code

J generate Java source code

h print help information

l *directory*
specify the directory where puma finds its data files

I *directory*
add directory to the list of directories to be searched for IMPORT files

FILES

<tree>.TS	description of the tree grammar
if output is in C:	
<module>.h	header file of the generated transformer module
<module>.c	body of the generated transformer module
yy<module>.h	macro file defining type specific operations
if output is in C++:	
<module>.h	header file of the generated transformer module
<module>.cxx	body of the generated transformer module
yy<module>.h	macro file defining type specific operations
if output is in Java:	
<module>.java	class of the generated transformer module
if output is in Modula-2:	
<module>.md	definition module of the generated transformer module
<module>.mi	implementation module of the generated transformer module

SEE ALSO

J. Grosch: "Puma - A Generator for the Transformation of Attributed Trees", CoCoLab Germany, Document No. 26

J. Grosch: "Transformation of Attributed Trees Using Pattern Matching", CoCoLab Germany, Document No. 27

Appendix 1: Syntax Summary

```
/* parser grammar */
Trafo           = Init Clauses Subroutines .

Init            = .

Clauses         = <
                = .
                = Clauses TRAFO DottedName .
                = Clauses TREE TreeNames .
                = Clauses PUBLIC ';' .
                = Clauses PUBLIC Names OptSemiColon .
                = Clauses IMPORT Name OptSemiColon .
                = Clauses EXTERN Names OptSemiColon .
                = Clauses EXPORT TargetBlock .
                = Clauses IMPORT TargetBlock .
                = Clauses GLOBAL TargetBlock .
                = Clauses BEGIN  TargetBlock .
                = Clauses CLOSE  TargetBlock .
                = Clauses GLOBAL OptSemiColon .
                = Clauses GLOBAL Declarations OptSemiColon .
> .
TreeNames       = <
                = .
                = TreeNames ',' .
                = TreeNames DottedName .
> .
ExternPart      = <
                = .
                = EXTERN Names ';' .
> .
Names           = <
                = .
                = Names ',' .
                = Names Name .
> .
Public          = <
                = .
                = PUBLIC .
> .
Subroutines     = <
                = .
                = Subroutines Public PROCEDURE Name '(' Parameters OutParameters ')'
                     ExternPart LocalCode LocalDecls Rules .
                = Subroutines Public FUNCTION Name '(' Parameters OutParameters ')'
                     Type ExternPart LocalCode LocalDecls Rules .
                = Subroutines Public PREDICATE Name '(' Parameters OutParameters ')'
                     ExternPart LocalCode LocalDecls Rules .
                = Subroutines Public TRIP Name '(' Parameters ')'
                     ExternPart LocalCode LocalDecls Rules .
> .
OutParameters   = <
                = .
                = '=>' Parameters .
> .
```

```
Parameters      = <
                = .
                = Mode Ident ':' Type .
                = Mode Type .
                = Mode Ident ':' Type ',' Parameters .
                = Mode Type ',' Parameters .
> .
Mode            = <
                = .
                = REF .
> .
Declarations    = <
                = Ident ':' Type .
                = Ident ':' Type AssignSymbol SimpleExpr .
                = Ident ':' Type ',' Declarations .
                = Ident ':' Type AssignSymbol SimpleExpr ',' Declarations .
> .
AssignSymbol    = <
                = ':=' .
                = '=' .
> .
Type            = <
                = Ident .
                = Ident '.' Name .
                = '[' Names ']' .
                = Ident '.' '[' Names ']' .
                = '{' TargetCodes '}' .
> .
LocalCode       = <
                = .
                = LOCAL TargetBlock .                                          |
> .
LocalDecls      = <
                = .
                = LOCAL ';' .                                                  |
                = LOCAL Declarations ';' .                                     |
> .
Rules           = <
                = .
                = Rules Patterns2 '.' .
                = Rules Patterns ':-' Statements '.' .
                = Rules Patterns '=>' Exprs2 '.' .
                = Rules Patterns Return1 '.' .                                 |
                = Rules Patterns '=>' Exprs ':-' Statements '.' .
                = Rules Patterns ':-' Statements '=>' Exprs2 '.' .
                = Rules Patterns '=>' Exprs Return1 '.' .                      |
                = Rules Patterns Return ':-' Statements '.' .                  |
                = Rules Patterns ':-' Statements Return2 '.' .                 |
                = Rules Patterns '=>' Exprs Return ':-' Statements '.' .       |
                = Rules Patterns '=>' Exprs ':-' Statements Return2 '.' .      |
                = Rules Patterns ':-' Statements '=>' Exprs Return1 '.' .      |
> .
OptSemiColon    = <
                = .
                = ';' .
> .
Return          = <                                                            |
```

```
                = RETURN SimpleExpr OptSemiColon .
                = RETURN OptSemiColon .
> .
Return1         = <
                = RETURN SimpleExpr ';' .
                = RETURN OptSemiColon .
> .
Return2         = <
                = RETURN SimpleExpr ';' Statements2 .
                = RETURN ';' Statements2 .
> .
Patterns        = <
                = Exprs .
                = Exprs ';' Patterns .
> .
Patterns2       = <
                = Exprs ';' .
                = Exprs ';' Patterns2 .
> .
Exprs           = <
                = '...' .
                = '...' ',' .
                = Expr .
                = Expr ',' Exprs .
                = NamedExprs .
> .
NamedExprs      = <
                = .
                = Ident ':=' Expr .
                = Ident ':=' Expr ',' NamedExprs .
> .
Exprs2          = <
                = '...' .
                = '...' ',' .
                = Expr ',' Exprs2 .
                = NamedExprs2 .
> .
NamedExprs2     = <
                = .
                = Ident ':=' Expr ',' NamedExprs2 .
> .
Exprs3          = <
                = Expr .
                = Expr ',' Exprs3 .
> .
Expr            = <
                = Expr2 .
                = Expr2 '?' SimpleExpr ':' SimpleExpr .
> .
Expr2           = <
                = PrefixExpr .
                = Expr2 Operator PrefixExpr .
                = Expr2 InOperator PrefixExpr .
> .
PrefixExpr      = <
                = Ident ':' PostfixExpr .
                = Ident ':>' PostfixExpr .
```

```
                = SimplePrefixExpr .
> .
PostfixExpr     = <
                = PrimaryExpr .
                = PostfixExpr '[' Exprs ']' .
                = PostfixExpr '(' Exprs ')' .
                = PostfixExpr '(' Exprs '=>' Exprs ')' .
                = PostfixExpr '.' Ident .
                = PostfixExpr '->' Ident .
                = PostfixExpr '^' .
                = PostfixExpr IncOperator .
> .
PrimaryExpr     = <
                = Ident .
                = NIL .
                = '_' .
                = Number .
                = Strings .
                = Ident '::' Ident .
                = '(' Expr ')' .
                = '(' Expr ',' Exprs3 ')' .
                = '(' Expr ')' PostfixExpr .
                = '(' Expr2 Operator ')' PostfixExpr .
> .
Strings         = <
                = String .
                = String Strings .
> .
SimpleExpr      = <
                = SimpleExpr2 .
                = SimpleExpr2 '?' SimpleExpr ':' SimpleExpr .
> .
SimpleExpr2     = <
                = SimplePrefixExpr .
                = SimpleExpr2 Operator SimplePrefixExpr .
                = SimpleExpr2 InOperator SimplePrefixExpr .
> .
SimplePrefixExpr= <
                = PostfixExpr .
                = Operator SimplePrefixExpr .
                = IncOperator SimplePrefixExpr .
                = GUARD SimplePrefixExpr .
                = '{' TargetCodes '}' .
> .
Statements      = <
                = .
                = Statements Statement .
                = Statements ';' .
> .
Statements2     = <
                = .
                = Statements2 Statement .
                = Statements2 RETURN SimpleExpr ';' .
                = Statements2 RETURN ';' .
                = Statements2 ';' .
> .
Statement       = <
```

```
                = SimpleExpr ';' .
                = SimpleExpr ':=' SimpleExpr ';' .
                = SimpleExpr AssignOperator SimpleExpr ';' .
                = SimpleExpr '?=' Expr ';' .
                = CONDITION SimpleExpr '?=' Expr ';' .
                = CONDITION SimpleExpr ';' .
                = IF SimpleExpr THEN Statements2 Elsifs END .
                = IF MatchConds THEN Statements2 Elsifs END .
                = FOR InitStmt ';' SimpleExpr ';' IncrStmt OptSemiColon
                     DO Statements2 END .
                = WHILE SimpleExpr DO Statements2 END .
                = FOR SimpleExpr DO Statements2 END .
                = FOR Ident ':' Type InOperator SimpleExpr DO Statements2 END .
                = REJECT .
                = FAIL .
                = Declarations ';' .
                = '{' TargetCodes '}' ';' .
                = NL .
> .
Elsifs          = <
                = .
                = ELSE Statements2 .
                = ELSIF SimpleExpr THEN Statements2 Elsifs .
                = ELSIF MatchConds THEN Statements2 Elsifs .
> .
MatchConds      = <
                = SimpleExpr '?=' Expr .
                = MatchConds ';' SimpleExpr '?=' Expr .
> .
InitStmt        = <
                = IncrStmt .
                = Ident ':' Type AssignSymbol SimpleExpr .
> .
IncrStmt        = <
                = .
                = SimpleExpr .
                = SimpleExpr ':=' SimpleExpr .
                = SimpleExpr AssignOperator SimpleExpr .
> .
TargetCodes     = <
                = .
                = TargetCodes Name Space '::' Space Ident .
                = TargetCodes Name Space '::' Space .
                = TargetCodes Name Space .
                = TargetCodes '::' .
                = TargetCodes TargetCode .
                = TargetCodes WhiteSpace .
> .
Name            = <
                = Ident .
                = String .
> .
DottedName      = <
                = Name .
                = DottedName '.' Name .
> .
Space           = <
```

```
                = .
                = Space WhiteSpace .
> .

/* lexical grammar */

Ident           : <
                = Letter .
                = '_' .
                = Ident Letter .
                = Ident Digit .
                = Ident '_' .
> .
Number          : <
                = Integer .
                = Real .
> .
Integer         : <
                = Digit .
                = Integer Digit .
> .
Real            : <
                = Integer '.' Integer Exponent .
                = Integer '.' Exponent .
                = '.' Integer Exponent .
> .
Exponent        : <
                = .
                = 'E' '+' Integer .
                = 'E' '-' Integer .
                = 'E' Integer .
> .
String          : <
                = "'" Characters "'" .
                = '"' Characters '"' .
> .
TargetBlock     : '{' Characters '}' .

TargetCode      : Characters .

WhiteSpace      : <
                = ' ' .
                = Tabulator .
                = Newline .
> .
Operator        : <
                = '!' .
                = '!=' .
                = '#' .
                = '%' .
                = '&' .
                = '&&' .
                = '*' .
                = '+' .
                = '-' .
                = '/' .
                = '<' .
                = '<<' .
                = '<=' .
```

```
                = '<>' .
                = '=' .
                = '==' .
                = '>' .
                = '>=' .
                = '>>' .
                = '|' .
                = '||' .
                = '~' .
                = AND .
                = DIV .
                = MOD .
                = NOT .
                = OR .
                = '\' Characters WhiteSpace .
> .
InOperator      : <                                                          |
                = IN .                                                       |
> .                                                                          |
IncOperator     : <
                = '++' .
                = '--' .
> .
AssignOperator  : <
                = '+=' .
                = '-=' .
                = '*=' .
                = '/=' .
                = '%=' .
                = '<<=' .
                = '>>=' .
                = '&=' .
                = '^=' .
                = '|=' .
> .
Comment         : '/*' Characters '*/' .
Characters      : <
                = .
                = Characters Character .
> .
/* replacements */
'...'           : < = '..'  .  > .
'<:'            : < = GUARD .  > .
```

Appendix 2: Examples from MiniLAX

The following examples are taken from a compiler for the demo language MiniLAX. The complete MiniLAX example can be found in [Grod]:

The first part contains the abstract syntax of the language and the output attributes which are assumed to be computed by a preceding semantic analysis phase. This information describes the structure of the input to a *puma* generated transformer. It is written in the input language of *ast*.

The second part specifies the generation of intermediate code. The abstract syntax tree is mapped to I-Code which is a subset of P-Code.

The third part specifies routines to handle types. Types are internally represented by trees. The routines are used by the semantic analysis phase which is implemented by an attribute grammar.

Appendix 2.1: Abstract Syntax

```
MODULE AbstractSyntax /* ---------------------------------------- */

TREE EXPORT  {
# include "Idents.h"
# include "Position.h"
}

GLOBAL  {
# include "Idents.h"
# include "Position.h"
# include <stdio.h>
}

EVAL Semantic

PROPERTY INPUT

RULE

MiniLAX         = Proc .
Decls           = <
   NoDecl       = .
   Decl         = Next: Decls REV [Ident: tIdent] [Pos: tPosition] <
      Var       = Type .
      Proc      = Formals Decls Stats .
   >.
>.
Formals         = <
   NoFormal     = .
   Formal       = Next: Formals REV [Ident: tIdent] [Pos: tPosition] Type .
>.
Type            = <
   Integer      = .
   Real         = .
   Boolean      = .
   Array        = Type OUT            [Lwb] [Upb] [Pos: tPosition] .
   Ref          = Type OUT .
   NoType       = .
   ErrorType    = .
>.
```

```
Stats           = <
   NoStat       = .
   Stat         = Next: Stats REV <
      Assign    = Adr Expr            [Pos: tPosition] .
      Call      = Actuals             [Ident: tIdent] [Pos: tPosition] .
      If        = Expr Then: Stats Else: Stats .
      While     = Expr Stats .
      Read      = Adr .
      Write     = Expr .
   >.
>.
Actuals         = <
   NoActual     =                     [Pos: tPosition OUT] .
   Actual       = Next: Actuals REV Expr .
>.
Expr            =                     [Pos: tPosition] <
   Binary       = Lop: Expr Rop: Expr [Operator: short] .
   Unary        = Expr                [Operator: short] .
   IntConst     =                     [Value         OUT] .
   RealConst    =                     [Value: double OUT] .
   BoolConst    =                     [Value: rbool  OUT] .
   Adr          = <
      Index     = Adr Expr .
      Ident     =                     [Ident: tIdent] .
   >.
>.
Coercions       = <
   NoCoercion   = .
   Coercion     = Next: Coercions OUT <
      Content   = .             /* fetch contents of location    */
      IntToReal = .             /* convert integer value to real */
   >.
>.

END AbstractSyntax

MODULE Output /* ------------------------------------------------ */

PROPERTY OUTPUT

DECLARE
   Formals Decls        = [Decls: tObjects THREAD] .
   Call Ident           = [Object: tObjects] [level: short] .
   If While             = [Label1] [Label2] .
   Read Write Binary    = [TypeCode: short] .
   Expr                 = Type Co: Coercions .
   Index                = type: Type .

END Output
```

Appendix 2.2: Generation of Intermediate Code

```
TRAFO ICode TREE Tree Defs PUBLIC Code

EXTERN
   ADD BoolType CHK ENT Emit EmitReal FJP FLT FalseCode INV IXA IntType JMP JSR
   LDA LDC LDI LES MST MUL REA RET RealType STI SUB TrueCode TypeSize WRI

GLOBAL {
# include "Tree.h"
# include "Defs.h"
# include "Types.h"
# include "ICodeInt.h"
}

PROCEDURE Code (t: Tree)

MiniLax (Proc) :-
        Code (Proc);
        .
Proc (Next := Next:Decls (Proc3 (ParSize := ParSize, DataSize := DataSize), ...),
                Decls := Decls, Stats := Stats) :-
        Emit (ENT, DataSize - ParSize, 0);
        Code (Stats);
        Emit (RET, 0, 0);
        Code (Decls);
        Code (Next);
        .
Var (Next := Next) :-
        Code (Next);
        .
Assign (Next, Adr, Expr, _) :-
        Code (Adr); Code (Adr::Co);
        Code (Expr); Code (Expr::Co);
        Emit (STI, 0, 0);
        Code (Next);
        .
Call (Next, Actuals, _, _, Proc3 (Level := Level, Label := Label,
                ParSize := ParSize), level) :-
        Emit (MST, level - Level, 0);
        Code (Actuals);
        Emit (JSR, ParSize - 3, Label);
        Code (Next);
        .
If (Next, Expr, Then, Else, Label1, Label2) :-
        Code (Expr); Code (Expr::Co);
        Emit (FJP, Label1, 0);
        Code (Then);
        Emit (JMP, Label2, 0);
        Code (Else);
        Code (Next);
        .
While (Next, Expr, Stats, Label1, Label2) :-
        Emit (JMP, Label2, 0);
        Code (Stats);
        Code (Expr); Code (Expr::Co);
```

```
        Emit (INV, 0, 0);
        Emit (FJP, Label1, 0);
        Code (Next);
        .
Read (Next, Adr, TypeCode) :-
        Code (Adr); Code (Adr::Co);
        Emit (REA, TypeCode, 0);
        Emit (STI, 0, 0);
        Code (Next);
        .
Write (Next, Expr, TypeCode) :-
        Code (Expr); Code (Expr::Co);
        Emit (WRI, TypeCode, 0);
        Code (Next);
        .
Actual (Next, Expr) :-
        Code (Expr); Code (Expr::Co);
        Code (Next);
        .
Binary (_, _, _, Lop, Rop, (Times}, TypeCode) :-
        Code (Lop); Code (Lop::Co);
        Code (Rop); Code (Rop::Co);
        Emit (MUL, TypeCode, 0);
        .
Binary (_, _, _, Lop, Rop, (Plus}, TypeCode) :-
        Code (Lop); Code (Lop::Co);
        Code (Rop); Code (Rop::Co);
        Emit (ADD, TypeCode, 0);
        .
Binary (_, _, _, Lop, Rop, (Less}, TypeCode) :-
        Code (Lop); Code (Lop::Co);
        Code (Rop); Code (Rop::Co);
        Emit (LES, TypeCode, 0);
        .
Unary (Expr := Expr) :-
        Code (Expr); Code (Expr::Co);
        Emit (INV, 0, 0);
        .
IntConst  (Value := Value  ) :- Emit (LDC, IntType, Value); .
RealConst (Value := Value  ) :- EmitReal (LDC, RealType, Value); .
BoolConst (Value := (true} ) :- Emit (LDC, BoolType, TrueCode); .
BoolConst (Value := (false}) :- Emit (LDC, BoolType, FalseCode); .

Index (_, _, _, Adr, Expr, Array (Type, Lwb, Upb, _)) :-
        Code (Adr); Code (Adr::Co);
        Code (Expr); Code (Expr::Co);
        Emit (CHK, Lwb, Upb);
        Emit (LDC, IntType, Lwb);
        Emit (SUB, IntType, 0);
        Emit (IXA, TypeSize (Type), 0);
        .
Ident (_, _, _, Ident, Var3 (Level := Level, Offset := Offset), level) :-
        Emit (LDA, level - Level, Offset);
        .
Content (Next) :-
        Emit (LDI, 0, 0);
        Code (Next);
```

```
        .
IntToReal (Next) :-
        Emit (FLT, 0, 0);
        Code (Next);
        .
```

Appendix 2.3: Procedures for Type Handling

```
TRAFO Types PUBLIC

Reduce                  /* return type without any ref levels          */

ReduceToRef             /* return type with ref level 1                */

Reduce1                 /* return type with 1 ref level removed        */

RefLevel                /* return number of ref levels of a type       */

IsSimpleType            /* check whether a type is simple              */

IsCompatible            /* check whether two types are compatible      */

IsAssignmentCompatible  /* check whether two types are                 */
                        /* assignment compatible                       */

ResultType              /* return the type of the result of            */
                        /* applying an operator to two operands        */

CheckParams             /* check a formal list of parameters           */
                        /* against an actual list of parameters        */

GetElementType          /* return the type of the elements of          */
                        /* an array type                               */

TypeSize                /* return the number of bytes used for         */
                        /* the internal representation of an           */
                        /* object of a certain type                    */

Coerce                  /* returns the coercion necessary to convert   */
                        /* an object of type 't1' to type 't2'         */

EXTERN nBoolean Error nNoCoercion

GLOBAL {
# include "Errors.h"
# include "Position.h"
# include "Tree.h"

# define Error(Text, Position) Message (Text, xxError, Position)

static tTree nBoolean, nNoType, nNoCoercion;
}

BEGIN {
   nBoolean     = mBoolean      ();
   nNoType      = mNoType       ();
   nNoCoercion  = mNoCoercion   ();
}

FUNCTION Reduce (Type) Type
   Ref (t)      RETURN Reduce (t) :- .
   t            RETURN t :- .

FUNCTION ReduceToRef (Type) Type
   Ref (t:Ref)  RETURN ReduceToRef (t) :- .
   t:Ref        RETURN t :- .
   t            RETURN t :- .

FUNCTION Reduce1 (Type) Type
   Ref (t)      RETURN t :- .
   t            RETURN t :- .
```

```
FUNCTION RefLevel (Type) int
   Ref (t)      RETURN RefLevel (t) + 1 :- .
   _            RETURN 0 :- .

PREDICATE IsSimpleType (Type)
   Array        :- FAIL; .
   _            :- .

PREDICATE IsCompatible (Type, Type)
   Integer    (), Integer    () :- .
   Real       (), Real       () :- .
   Boolean    (), Boolean    () :- .
   Array (t1, Lwb, Upb, _), Array (t2, Lwb, Upb, _) ;
   Ref      (t1), t2             ;
   t1           , Ref      (t2) :- IsCompatible (t1, t2); .
   NoType     (), _              :- .
   _            , NoType     () :- .
   ErrorType  (), _              :- .
   _            , ErrorType  () :- .

PREDICATE IsAssignmentCompatible (Type, Type)
   Integer    (), Integer    () :- .
   Real       (), Real       () :- .
   Real       (), Integer    () :- .
   Boolean    (), Boolean    () :- .
   Ref (t1)     , t2             ;
   t1           , Ref (t2)       :- IsAssignmentCompatible (t1, t2); .
   NoType     (), _              :- .
   _            , NoType     () :- .
   ErrorType  (), _              :- .
   _            , ErrorType  () :- .

FUNCTION ResultType (Type, Type, int) Type
   t:Integer  (), Integer     (), (Plus )   RETURN t           :- .
   t:Real     (), Real        (), (Plus )   RETURN t           :- .
   t:Integer  (), Integer     (), (Times)   RETURN t           :- .
   t:Real     (), Real        (), (Times)   RETURN t           :- .
   Integer    (), Integer     (), (Less )   RETURN nBoolean    :- .
   Real       (), Real        (), (Less )   RETURN nBoolean    :- .
   t:Boolean  (), Boolean     (), (Less )   RETURN t           :- .
   t:Boolean  (), _             , (Not  )   RETURN t           :- .
   Ref (t1)     , t2            , o         ;
   t1           , Ref (t2)      , o         RETURN ResultType (t1, t2, o) :- .
   t:NoType   (), _             , _         RETURN t           :- .
   _            , t:NoType    (), _         RETURN t           :- .
   ErrorType  (), _             , _         RETURN NoType    () :- .
   _            , ErrorType   (), _         RETURN NoType    () :- .
   ...                                      RETURN ErrorType () :- .

PROCEDURE CheckParams (Actuals, Formals)
   NoActual (...), NoFormal ()  :- .
   NoActual (Pos), _            :-
      Error ("too few actual parameters"        , Pos); .
   Actual (_, Expr (Pos, ...)), NoFormal () :-
      Error ("too many actual parameters"       , Pos); .

/* alternative 1 */
```

```
   Actual (NextA, Expr (Pos, TypeA, ...)), Formal (_, _, NextF, _, _, TypeF) :-
      {
         if (! IsCompatible (TypeA, TypeF))
            Error ("parameter type incompatible", Pos);
         if (! (RefLevel (TypeF) - 1 <= RefLevel (TypeA)))
            Error ("variable required"          , Pos);
      };
      CheckParams (NextA, NextF); .

/* alternative 2 */

   Actual (NextA, Expr (Pos, TypeA, ...)), Formal (_, _, NextF, _, _, TypeF) :-
      ! IsCompatible (TypeA, TypeF);
      Error ("parameter type incompatible"      , Pos);
      REJECT; .

   Actual (NextA, Expr (Pos, TypeA, ...)), Formal (_, _, NextF, _, _, TypeF) :-
      ! (RefLevel (TypeF) - 1 <= RefLevel (TypeA));
      Error ("variable required"                , Pos);
      REJECT; .

   Actual (NextA, Expr (Pos, TypeA, ...)), Formal (_, _, NextF, _, _, TypeF) :-
      CheckParams (NextA, NextF); .

/* alternative 3 */

   Actual (NextA, Expr (Pos, TypeA, ...)), Formal (_, _, NextF, _, _, TypeF) :-
      CheckCompatible (Pos, TypeA, TypeF);
      CheckRefLevel (Pos, TypeA, TypeF);
      CheckParams (NextA, NextF); .

PROCEDURE CheckCompatible (tPosition, Type, Type)
   _    , t1    , t2    :- IsCompatible (t1, t2); .
   Pos  , ...           :- Error ("parameter type incompatible"  , Pos); .

PROCEDURE CheckRefLevel (tPosition, Type, Type)
   _    , t1    , t2    :- RefLevel (t2) - 1 <= RefLevel (t1); .
   Pos  , ...           :- Error ("variable required"           , Pos); .

FUNCTION GetElementType (Type) Type
   Array (t, ...)       RETURN t :- .
   _                    RETURN NoType () :- .

FUNCTION TypeSize (Type) int
   Array (t, Lwb, Upb, _)       RETURN (Upb - Lwb + 1) * TypeSize (t) :- .
   _                            RETURN 1 :- .

FUNCTION Coerce (t1: Type, t2: Type) Coercions
   Ref     (T1) , Ref  (T2)     RETURN Coerce (T1, T2) :- .
   Integer ()   , Real ()       RETURN IntToReal (nNoCoercion) :- .
   Ref     (T1) , T2            RETURN Content (Coerce (T1, T2)) :- .
   ...                          RETURN nNoCoercion :- .
```

Appendix 3: Equality Operations

Appendix 3.1: C

```
# define equalint(a, b)         (a) == (b)
# define equalshort(a, b)       (a) == (b)
# define equallong(a, b)        (a) == (b)
# define equalunsigned(a, b)    (a) == (b)
# define equalfloat(a, b)       (a) == (b)
# define equaldouble(a, b)      (a) == (b)
# define equalrbool(a, b)       (a) == (b)
# define equalchar(a, b)        (a) == (b)
# define equalvchar(a, b)       (a) == (b)
# define equaltString(a, b)     strcmp (a, (b)) == 0
# define equaltStringRef(a, b)  (a) == (b)
# define equaltWStringRef(a, b) (a) == (b)
# define equalvtStringRef(a, b) (a) == (b)
# define equaltIdent(a, b)      (a) == (b)
# define equaltWIdent(a, b)     (a) == (b)
# define equalvtIdent(a, b)     (a) == (b)
# define equaltSet(a, b)        IsEqual (& a, & b)
# define equaltPosition(a, b)   Compare (a, b) == 0
# define equalNodeHead(a, b)    rtrue
```

Appendix 3.2: C++

```
# define equalint(a, b)         (a) == (b)
# define equalshort(a, b)       (a) == (b)
# define equallong(a, b)        (a) == (b)
# define equalunsigned(a, b)    (a) == (b)
# define equalfloat(a, b)       (a) == (b)
# define equaldouble(a, b)      (a) == (b)
# define equalrbool(a, b)       (a) == (b)
# define equalbool(a, b)        (a) == (b)
# define equalchar(a, b)        (a) == (b)
# define equalvchar(a, b)       (a) == (b
# define equaltString(a, b)     strcmp (a, (b)) == 0
# define equaltStringRef(a, b)  (a) == (b)
# define equaltWStringRef(a, b) (a) == (b)
# define equalvtStringRef(a, b) (a) == (b)
# define equaltIdent(a, b)      (a) == (b)
# define equaltWIdent(a, b)     (a) == (b)
# define equalvtIdent(a, b)     (a) == (b)
# define equaltSet(a, b)        IsEqual (& a, & b)
# define equaltPosition(a, b)   Compare (a, b) == 0
# define equalNodeHead(a, b)    true
```

Appendix 3.3: Java

```
# define equalboolean(a, b)     (a) == (b)
# define equalbyte(a, b)        (a) == (b)
# define equalchar(a, b)        (a) == (b)
# define equaldouble(a, b)      (a) == (b)
# define equalfloat(a, b)       (a) == (b)
# define equalint(a, b)         (a) == (b)
# define equallong(a, b)        (a) == (b)
```

```
# define equalshort(a, b)       (a) == (b)
# define equalIdent(a, b)       (a.equals (b))
# define equalPosition(a, b)    (a.compareTo (b) == 0)
# define equalNodeHead(a, b)    true
```

Appendix 3.4: Modula-2

```
# define equalINTEGER(a, b)     (a) = (b)
# define equalSHORTINT(a, b)    (a) = (b)
# define equalLONGINT(a, b)     (a) = (b)
# define equalCARDINAL(a, b)    (a) = (b)
# define equalSHORTCARD(a, b)   (a) = (b)
# define equalLONGCARD(a, b)    (a) = (b)
# define equalREAL(a, b)        (a) = (b)
# define equalLONGREAL(a, b)    (a) = (b)
# define equalBOOLEAN(a, b)     (a) = (b)
# define equalCHAR(a, b)        (a) = (b)
# define equalWCHAR(a, b)       (a) = (b)
# define equalBITSET(a, b)      (a) = (b)
# define equalBYTE(a, b)        (a) = (b)
# define equalWORD(a, b)        (a) = (b)
# define equalADDRESS(a, b)     (a) = (b)
# define equaltString(a, b)     Strings.IsEqual (a, b)
# define equaltWString(a, b)    WStrings.IsEqual (a, b)
# define equaltStringRef(a, b)  (a) = (b)
# define equaltWStringRef(a, b) (a) = (b)
# define equaltIdent(a, b)      (a) = (b)
# define equaltWIdent(a, b)     (a) = (b)
# define equaltText(a, b)       FALSE
# define equaltSet(a, b)        Sets.IsEqual (a, b)
# define equaltRelation(a, b)   Relation.IsEqual (a, b)
# define equaltPosition(a, b)   Position.Compare (a, b) = 0
# define equalNodeHead(a, b)    TRUE
```

References

[Groa] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Cocktail Document No. 15, CoCoLab Germany.

[Grob] J. Grosch, Reusable Software - A Collection of Modula-Modules, Cocktail Document No. 4, CoCoLab Germany.

[Groc] J. Grosch, Reusable Software - A Collection of C-Modules, Cocktail Document No. 30, CoCoLab Germany.

[Grod] J. Grosch, Specification of a Minilax Interpreter, Cocktail Document No. 22, CoCoLab Germany.

Contents

1.	Introduction	1
2.	Overview	2
3.	Input Language	3
3.1.	Notation	3
3.2.	Lexical Conventions	3
3.3.	Structure	4
3.4.	Target Code	5
3.5.	Subroutines	6
3.5.1.	Trip	7
3.6.	Types	8
3.7.	Rules	8
3.8.	Patterns	10
3.9.	Expressions	11
3.10.	Statements	13
3.11.	Equality Operations	17
3.12.	Begin Operations	18
3.13.	Output Statements	19
4.	Scopes	20
5.	Output	20
5.1.	C++	21
5.1.1.	Simple C++	21
5.1.2.	Proper C++	21
5.1.3.	Idents_PREFIX	21
6.	Usage	22
	Appendix 1: Syntax Summary	25
	Appendix 2: Examples from MiniLAX	32
	Appendix 2.1: Abstract Syntax	32
	Appendix 2.2: Generation of Intermediate Code	34
	Appendix 2.3: Procedures for Type Handling	37
	Appendix 3: Equality Operations	40
	Appendix 3.1: C	40
	Appendix 3.2: C++	40
	Appendix 3.3: Java	40
	Appendix 3.4: Modula-2	41
	References	41